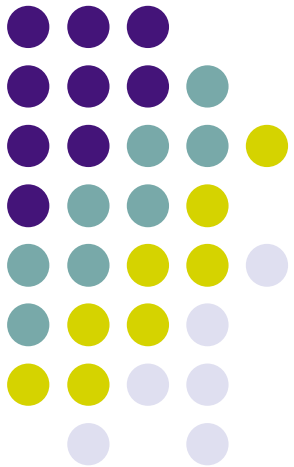
OOP: Type-bound Procedures and Inheritance in Fortran 2003

Tom Clune

SIVO Fortran 2003 Series

May 6, 2008



Logistics

- Materials for this series can be found at http://modelingguru.nasa.gov/clearspace/docs/DOC-1375
  - Contains slides and source code examples.
  - Latest materials may only be ready at-the-last-minute.
- Please be courteous:
  - Remote attendees should use “*6” to toggle the mute. This will minimize background noise for other attendees.

Outline

- Variables as Objects
  - Type-bound procedures
  - Invocation of methods
  - Operators
  - Constructors
  - Final procedures
- Type Extension and Inheritance
  - Extends attribute
  - Abstract Types
  - Abstract Interfaces
  - Deferred procedures

Type-bound Procedures

- Or “Procedures Bound to a Type by Name”
- Allows a Fortran subroutine or procedure to be treated as a *method* on an object of the given type.

```
call object % method(…)
x = object % func(…)
```

  which is equivalent to

```
call method(object, …)
x = func(object, …)
```

- This syntax encourages an object-oriented style of programming that can improve clarity in some contexts.
- Procedures can also be bound by type to operators: [=, +, etc.]
- Restrictions
  - *Cannot* be used with SEQUENCE types.

Type-bound Syntax

- The following example defines a derived type with 2 type bound procedures `compute()` and `retrieve()`

```
type myType
   real :: value
contains
   procedure :: compute
   procedure :: retrieve
end type myType
```

- Type-bound procedures must be module procedures or external procedures with explicit interfaces.
- By default type-bound names are public, but each entity in a type (including components) can be have a `PUBLIC/PRIVATE` attribute.

Type-bound Procedures

- Specific type-bound procedures
  - Syntax

    `procedure [ (`*`interface-name`*`) ] [ [,`*`bind-attr-list`*` ] ::`
    `] `*`tbp-name`*` [=>`*`procedure-name`*`]`

    where *`bind-attr-list`* is one of

    `access-spec` (`public` or `private`)

    `deferred`

    `nopass`

    `non_overridable`

    `pass [(`*`arg-name`*`) ]`
  - `(`*`interface-name`*`)` if and only if `deferred` attribute
  - `non_overridable` => cannot be overridden during type extension

Full example

```
module myType_mod
    private
    public :: myType
    type myType
         real :: myValue(4) = 0.0
    contains
         procedure :: write
         procedure :: reset
    end type myType
contains
    subroutine write (this, unit)
       class(mytype) :: this
       integer, optional :: unit
       if(present(unit)) then
          write (unit, *) this % myValue
       else
          print *, this%myvalue
        endif
    end subroutine write_mytype
…
```

```
…
    subroutine reset(variable)
       class(mytype) :: variable
       variable%myvalue = 0.0
    end subroutine reset
end module myType_mod
```

Usage:

```
use mytype_mod

type (myType) :: var

…

call var%write(unit=6)

call var % reset()
```

Passed-object dummy arg

- By default, type-bound procedures pass the object as the first argument.
  - Can override behavior with **NOPASS** attribute:

```
procedure, NOPASS :: method
…
call thing % method(…) ! No object is passed
```

  - Can also specify which argument is to be associated with the passed-object with the **PASS** attribute:

```
procedure, PASS(obj) :: method
…
subroutine method(x,obj,y)
…
call thing % method(x, y) ! Thing is 2nd obj.
```

  - The default can be explicitly confirmed by

```
procedure, PASS :: method
```

- ***Strongly*** recommend that you always use the default.

Renaming and Generic

- Type-bound procedures can specify an alternative public name using a mechanism analagous to that for the module `ONLY` clause:

```
procedure :: write => writeInternal
```

- Similarly, an external name can be overloaded for multiple interfaces with the GENERIC statement:

```
type myType
contains
   procedure :: addInteger
   procedure :: addReal
   generic ::add=>addInteger,addReal
end type
```

Type-bound Operators

- Syntax is through the GENERIC statement:

```
type SomeType
   …
contains
   GENERIC :: OPERATOR(+) => myAdd
   GENERIC :: OPERATOR(=) => myAssign
end type

type (SomeType) :: a, b, c
…
a = b + c ! Invokes myAdd, then myAssign
```

Structure Constructors

- To support object oriented programming, F2003 has introduced more general structure constructors
- A generic name may the same as a derived type name
  - Allows standard OO style where class constructors use the same name as that of the class itself.
  - Default constructor accepts an argument for each public component
    - Requires any/all private components to have default initial values.
  - User may overload generic name to provide multiple means for constructing objects of a given type.

Finalizers

- A derived type may have '*final*' subroutines bound to it.
  - Designated with **FINAL** keyword

```
type T
  contains
    FINAL :: cleanUp
  end type
```

  - Intent is to clean-up when objects cease to exist (I.e. go out of scope).
    - Usually this involves deallocating components of a derived type
    - Potentially eliminates issues related to memory leaks.
- Note: FINAL routines are **always** available regardless of PUBLIC/PRIVATE designation
    - **Invoked automatically *not* by user code!**
    - Can catch developers off guard.

Finalizers cont'd

- Sole dummy argument cannot be INTENT(OUT)
- Can overload for multiple ranks - or use ELEMENTAL
- A derived type is *finalizable* if it has any final subroutines ***or*** has components that are finalizable
  - Final subroutines only deal with those components which are not themselves finalizable.

Inheritance

- Fortran 2003 introduces OOP *inheritance* via the `EXTENDS` attribute for user defined types.
  - Implementation is restricted to *single* inheritance
    - Inheritance always forms hierarchical trees.
    - No “diamond” patterns which can happen in C++
  - Implementation is designed to be efficient such that offsets for components and type-bound procedures can be computed at compile time. (“single lookup”)
- With type extension, a developer may add new components and type-bound procedures to an existing derived type even *without* access to the source code for that type.

Inheritance terminology

- A type is considered to be *extensible* if it is:
  - Not a SEQUENCE type
  - Not BIND(C)
- An extensible type *without* the EXTENDS attribute is considered to be a '*base type*'.
  - Base types need not have *any* components.
  - Extension need not *add* any components.
- A type *with* the EXTENDS attribute is said to be an *extended* type.
  - '*Parent type*' is used for the type from which the extension is made.
  - All the components, and bound procedures of the parent type are inherited by the extended type and they are known by the same names.
  - An extended type can be a parent for yet another type, and so on.

Syntax for Extends

```
type Location2D
  real :: latitude, longitude
end type Location2D

type, EXTENDS (Location2D) :: Location3D
  real :: pressureHeight
end type Location3D
…
type (Location3D) :: location
lat = location % latitude
lon = location % longitude
height = location % pressureHeight
```

Parent type

The Parent Component

- Every extended type has an implicit component associated with the parent type
  - The component name is the type name of the parent.
  - Provides multiple mechanisms to access components in parent type
- From the previous example we could do:

```
type(Location2D) :: latLon
latLon = location % Location2D
lat = location % Location2D % latitude
```

Extends and Type-bound

- Type-bound procedures in the parent may be invoked within extended types.
- Extended types may add additional type-bound procedures in the natural fashion.
- An extended type can *override* a type-bound procedure in the parent - specifying new behavior in the extended type.
  - The keyword **`NON_OVERRIDABLE`** can be used to prohibit extended classes from overriding behavior:

```
procedure, NON_OVERRIDABLE :: foo
```

Overriding Example

```
type Square
   real :: length
contains
   procedure :: area => squareArea
end type Square

type, extends(Square) :: rectangle
   real :: width
contains
   procedure :: area => rectangleArea
end type Rectangler

Real function squareArea(this)
   squareArea = (this % length) ** 2

Real function rectangleArea(this)
   rectangleArea = (this % length) * (this % width)
```

Abstract Types

- It is often useful to have a base type that declares methods (type-bound procedures) that are not implemented except in extended classes.
- Fortran 2003 uses the **`ABSTRACT`** attribute to denote such a type.
  - The **`DEFERRED`** attribute is used for those methods which are not to be implemented.
    - Requires a template or abstract interfac
  - No variables can be declared to be of an abstract type.

Abstract Example

```
type, ABSTRACT :: AbstractShape
contains
  procedure (AreaInterface), DEFERRED :: area
end type AbstractShape
…
abstract interface
  subroutine AreaInterface(obj)
     import AbstractShape
     class (AbstractShape) :: obj
  end subroutine AreaInterface
end interface
```

Abstract Example cont'd

```
type, extends(AbstractShape) :: Square
   real :: length
contains
  procedure :: area => squareArea !
  Provide concrete
end type Square
```

Supported features

- Generaly, IBM XLF and NAG F95 support the majority of the features presented here.
- g95, gfortran generally support none of these yet.

Pitfalls and Best Practices

- Type-bound procedures:
  - Always use the first argument (the default) as the object argument for type-bound procedures.
  - Use a standardized name for the object argument such as “this” or “self” to increase clarity.

Resources

- **SIVO Fortran 2003 series:** https://modelingguru.nasa.gov/clearspace/docs/DOC-1390
- **Questions to Modeling Guru:** https://modelingguru.nasa.gov
- **SIVO code examples on Modeling Guru**
- **Fortran 2003 standard:** http://www.open-std.org/jtc1/sc22/open/n3661.pdf
- ***John Reid summary:***
  - ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.pdf
  - ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.ps.gz
- ***Newsgroups***
  - http://groups.google.com/group/comp.lang.fortran
- ***Mailing list***
  - http://www.jiscmail.ac.uk/lists/comp-fortran-90.html

Next Fortran 2003 Session

- OOP: Polymorphism in Fortran 2003
- Tom Clune will present
- Tuesday, May 20, 2008
- B28-E210 @ 12:00 noon